Duqu contre Duqu : Analyse et détournement du driver de Duqu

Guillaume Bonfante, Jean-Yves Marion, Fabrice Sabatier et Aurélien Thierry
prenom.nom@loria.fr

LORIA, Inria, Université de Lorraine

Résumé Proche de `Stuxnet` avec lequel il partage des fonctionnalités, `Duqu` a fait l'objet de nombreuses analyses au cours de l'année passée. Nous nous proposons d'en étudier un élément en particulier, le driver. Notre contribution consiste en la rétroingénierie de ce composant par la reconstitution de son code source et une analyse de ses fonctionnalités. Nous avons ensuite détourné le driver pour détecter d'éventuelles injections dans les exécutables Windows et prévenir d'autres attaques. En particulier nous montrons comment le driver de `Duqu` modifié aurait permis de détecter `Duqu`.

1 Introduction

Lors de sa découverte en septembre 2011 par `CrySys`, laboratoire de sécurité et de cryptographie de l'université de Budapest [6], il a été dit de `Duqu`, un malware d'espionnage, qu'il était apparenté à un autre code malveillant : `Stuxnet`. L'argument tient au fait que les deux malware emploient les *mêmes* mécanismes pour assurer leur primo-infection ; or, la sophistication même de ces mécanismes est une signature de leurs auteurs, et donc lie les deux attaques. Mais l'adage veut que les systèmes complexes soient fragiles. Nous illustrons ici ce principe en détournant les fonctionnalités de `Duqu` pour en faire un système de détection d'injection de code, rudimentaire mais capable de détecter une attaque par Duqu.

`Duqu` est un outil offensif utilisé pour le vol d'informations. Symantec [9] a identifié parmi les fonctionnalités de `Duqu` des enregistreurs de frappes (*keylogger*), de l'écran, de l'activité réseau, des programmes ouverts, ainsi des outils de découverte de services. Le malware est maintenu à jour via un serveur de Command & Control. Il est également muni d'un mécanisme d'auto-destruction après 36 jours s'il n'a pas reçu de notification contraire du C&C. Les attaques semblent réussies puisque le malware n'a pas été détecté à chaud alors que certaines opérations ont duré plusieurs mois, mais seulement "post-mortem". De nombreuses souches du malware ont été trouvées dans la nature, chacune avec des binaires différents mais

similaires. Kasperky a fait un bon historique des versions [7], la dernière souche détectée datant de février 2012 alors que `Duqu` était déjà largement découvert et documenté. En résumé, `Duqu` est un logiciel espion, discret, s'adaptant spécifiquement à sa cible.

Or, nous développons au Laboratoire de Haute Sécurité du LORIA, un prototype de détecteur de virus fondé sur l'analyse morphologique [4], technique basée sur la comparaison des graphes de flot de contrôle. Dès que nous avons eu les échantillons de `Stuxnet` et `Duqu`, nous avons cherché à savoir si notre méthode –connaissant `Stuxnet`– aurait permis de détecter l'attaque de `Duqu`. De fait, la réponse est *positive* si nous disposons d'un exemplaire déchiffré de la DLL principale de `Duqu`. Jusque là, notre système ne procédait pas à un tel déchiffrement de manière automatique. Nous montrons dans cet article que certains mécanismes du driver d'infection de `Duqu` peuvent être réemployés pour obtenir une telle version déchiffrée du malware au cours de l'attaque. C'est donc à l'aide d'une version remaniée de `Duqu` que notre détecteur identifie `Duqu`.

Nous y revenons en détail par la suite, mais disons quelques mots sur le procédé d'infection de `Duqu`. En ouvrant un document WORD exploitant une vulnérabilité du noyau, l'utilisateur provoque l'installation sur la machine cible d'un *driver* et de fichiers annexes *chiffrés*. Après reboot, le driver observe les processus chargés en mémoire par le système. Il s'intéresse en particulier au processus `services.exe` qu'il infecte lorsqu'il est chargé en mémoire. C'est alors `services.exe`, modifié, qui utilise la DLL de `Duqu` et met l'attaque en place.

Partant de là, le scénario de contre-attaque est clair : il consiste à modifier le driver de `Duqu` de telle sorte qu'il puisse observer le chargement en mémoire des processus, les extraire après déchiffrement éventuel et leur faire passer l'analyse morphologique. Mais de la théorie à la pratique, il y a une marche. Pour mettre en œuvre ce plan, nous avons

- Reconstitué, par rétroingénierie, le code source du driver de `Duqu` à partir de son binaire,
- Modifié le code pour surveiller le chargement des processus par le système,
- Interfacé le nouveau driver avec notre outil de détection.

Nous reviendrons dans les sections qui suivent sur chacune de ces étapes. Nous en profiterons pour revenir sur une analyse détaillée des techniques utilisées par `Duqu`. Nous allons, dans cette introduction, rappeler le scénario d'une infection par `Duqu` et sa technique de propagation. Puis nous détaillerons la démarche effectuée pour reconstruire le code du driver à partir du fichier binaire et nous ferons une analyse fonctionnelle

de celui-ci à l'aide du code source. Enfin nous proposons une application du driver à la détection de malware en surveillant les DLL utilisées par le système.

Déroulement de l'infection L'infection détectée par CrySyS utilisait un document Microsoft Word piégé incluant `Duqu`. Dans un premier temps, le document word utilise une faille 0-day du noyau (faille sur les polices de caractères TrueType [1]), déchiffre et installe trois composants : un driver (`nfrd965.sys`), une DLL (la DLL principale de `Duqu`, `NETP191.PNF`) , et un fichier de configuration (`netp192.PNF`).

La seconde étape se déroule au redémarrage de la cible. Le driver injecte la DLL de `Duqu` dans un processus spécifié par le fichier de configuration, typiquement `services.exe`.

Dans la troisième (et dernière) étape, la charge finale, présente dans la DLL, est installée.

Une des difficultés pour détecter le malware est que seule la DLL et le driver sont actifs sur le système. La DLL est chiffrée, packée avec UPX et elle est injectée dès qu'elle est déchiffrée mais elle n'apparaît donc déchiffrée qu'en RAM. Le moment où l'injection est réalisée, soit une fois que la DLL est déchiffrée mais avant qu'elle ne soit activée, est donc l'instant privilégié pour analyser la DLL et détecter l'attaque.

Une fois installé sur une première machine cible, `Duqu` utilise un schéma de réplication contrôlé par un attaquant humain permettant une propagation discrète au sein d'un réseau protégé. Chaque nouvelle machine infectée peut-être configurée pour se connecter à l'attaquant non pas directement mais par la machine qui l'a infectée, créant une sorte de tunnel de routage pour des machines non accessibles directement depuis l'extérieur (Figure 1).

2 Reconstruction du code du driver

Nous savions que `Duqu` partage du code avec Stuxnet puisque nous avons exhibé des parties équivalentes dans les DLL principales de chacun des malware [11] et Symantec a détaillé des similarités dans la technique d'injection de la DLL et dans certaines ressources (exports et routines). Pourquoi alors s'intéresser au driver de `Duqu` alors que celui de `Stuxnet`, similaire, a été décompilé par Amr Thabet [10] ? En réalité le diable est dans les détails car l'analyse détaillée d'une souche du driver découverte en octobre 2011 en Europe (`nfrd965.sys`) a montré de nombreuses singularités par rapport à `Stuxnet` dans sa méthode d'injection et ses mécanismes de furtivité.

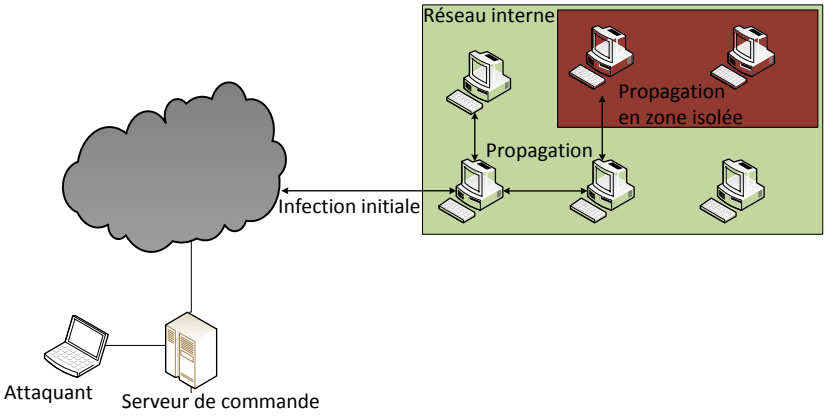

Figure 1. Schéma de propagation en profondeur de Duqu

Nous avons donc travaillé sur la rétroingénierie de cette version spécifique du driver afin d'en documenter les fonctionnalités. Notre objectif est d'obtenir un code compréhensible, qui compile, et dont la version compilée soit au plus proche du driver original.

2.1 Décompilation avec IDA

Nous avons utilisé le module de décompilation "Hex-Rays Decompiler", intégré à IDA sous la forme d'un plugin [2]. Il permet de générer un pseudo-code C à partir du fichier binaire en cours d'analyse. Il produit non seulement du code source mais facilite également sa réécriture directement à l'intérieur de l'interface graphique du plugin. Malheureusement le code en sortie n'est, dans notre cas, pas exploitable directement. D'une part le code n'est pas compilable parce que des types de variables n'ont pas été correctement reconnus et certaines conventions d'appel ne sont pas standard (non reconnues par le décompilateur). De plus le code généré est difficilement lisible, en partie parce que certaines structures n'ont pas été identifiées. Nous détaillerons dans les paragraphes suivants ces difficultés et des moyens de résolution.

Nous avons procédé de manière incrémentale afin de reconstruire le code petit à petit en vérifiant à chaque étape que le code compile et qu'une fois compilé il est équivalent à celui du binaire `nfrd965.sys` original. Cela a consisté à :

- Commenter tout le pseudo-code sauf la fonction du point d'entrée du driver (*DriverEntry*) et les variables globales s'y rapportant
- Régler chaque erreur une par une

– Comparer le code compilé au binaire original, modifier le code pour s'en rapprocher
– Ajouter du code auparavant commenté et revenir à l'étape de correction d'erreurs.

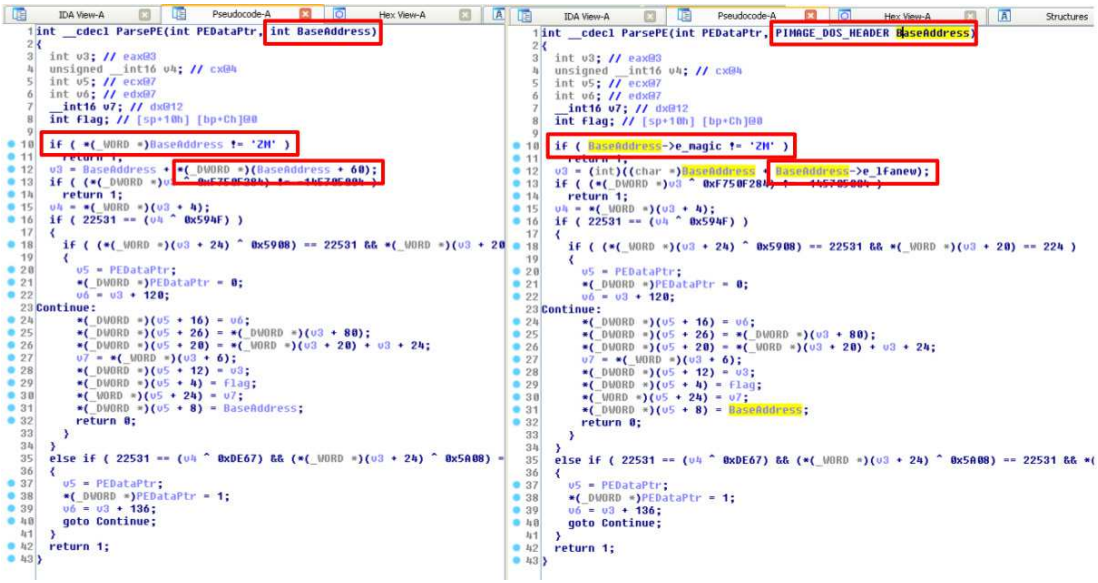

Figure 2. À gauche, le code source initialement généré par IDA. À droite, une fois le type de la variable `BaseAddress` re-défini.

2.2 Identification des structures et des types

Comme on peut s'en rendre compte sur la partie gauche de la figure 2, le décompilateur n'arrive pas toujours à déterminer le type d'une variable. On prend l'exemple de la fonction `ParsePE`, utilisée pour chercher des informations dans les binaires chargés en mémoire. On part donc à la pêche aux indices : ici le test sur la valeur magique `"MZ"` nous laisse à penser que la fonction porte sur un traitement de l'entête d'un fichier exécutable au format PE, ce qui nous permet d'en déduire que la variable `BaseAddress` est un pointeur sur une structure `IMAGE_DOS_HEADER`. Une fois le type de la variable correctement défini, IDA identifie automatiquement le nom du champ de la structure à l'*offset* indiqué nous épargnant de longues recherches dans la documentation de Microsoft (par exemple l'*offset* 60 soit 0x3C en hexadécimal correspond au champ `e_lfanew` de la structure `IMAGE_DOS_HEADER`). Si la structure est propre au programme analysé, nous pouvons la définir à travers le plugin de décompilation après une analyse manuelle.

Au final, nous avons retouché le code pour obtenir un code compréhensible pour un développeur C (figure 3). Ce n'est pas encore un code "propre" : par exemple un développeur n'aurait pas utilisé de *goto* pour gérer un traitement commun à la fin d'une condition mais l'aurait plutôt placé après la disjonction des cas, et il aurait rajouté une clause *else* pour le cas par défaut. Après cette réorganisation, les objectifs sont atteints : pour la fonction `ParsePE`, le code (re)compilé est proche du binaire original, et le code source reconstruit est exploitable.

La lisibilité du code ainsi obtenu nous permet de détecter une erreur de programmation comme à la deuxième ligne ici :

```
else if ((pFileHeader->Machine ^ 0xDE67) == (IMAGE_PE_x86_MACHINE ^ 0xDE67)
         && (pOptionHeader->Magic ^ 0x5A08) == IMAGE_PE32_PLUS_MAGIC
         && pFileHeader->SizeOfOptionalHeader == 0xF0 ) // 0x5803 64bits
```

Au lieu de :

```
else if ((pFileHeader->Machine ^ 0xDE67) == (IMAGE_PE_x86_MACHINE ^ 0xDE67)
         && (pOptionHeader->Magic ^ 0x5A08) == (IMAGE_PE32_PLUS_MAGIC ^ 0
             x5A08)
         && pFileHeader->SizeOfOptionalHeader == 0xF0 ) // 0x5803 64bits
```

Le test mis en place omet un XOR sur la constante IMAGE_PE32_PLUS_MAGIC. Ce test devrait permettre de reconnaître un binaire 64 bits et de définir une table d'import en conséquence. Avec cette erreur, `Duqu` est incapable de parser un binaire 64 bits (un code d'erreur est renvoyé). De fait cette version du driver ne fonctionne pas sur un Windows 64 bits.

2.3 Conventions d'appel

Pour chaque routine, IDA cherche à déterminer la convention d'appel utilisée à partir des registres qui sont lus avant d'être écrits (paramêtres) et ceux écrits sans être lus après (valeur de retour). Si ces registres correspondent à un appel classique, il l'annote dans le code C pour que le compilateur respecte la convention. Les conventions d'appel de Microsoft Visual C++ sont données Figure 4, l'appel par défaut étant *thiscall*. Dans le cas où il ne détermine pas la convention, il annote les registres d'entrée et de sortie en notant qu'il s'agit d'un appel non conventionnel (*usercall*) et met la définition de la fonction en commentaire (ici les arguments sont passés dans les registres `edi` et `esi`) :

```
NTSTATUS __cdecl ParsePE(
      __out PEDataPtr pPEData,
      __in PIMAGE_DOS_HEADER BaseAddress,
      __in int flag)
{
  PVOID infosPE; //IMAGE_NT_HEADER;
  PIMAGE_DOS_HEADER  pDosHeader     = NULL;
  PIMAGE_NT_HEADERS  pNtHeader      = NULL;
  PIMAGE_FILE_HEADER  pFileHeader   = NULL;
  PIMAGE_OPTIONAL_HEADER32  pOptionHeader = NULL;
  PVOID pExportTableRVA;

  infosPE=(PIMAGE_DOS_HEADER)BaseAddress;
  if (((PIMAGE_DOS_HEADER)infosPE)->e_magic != 'ZM' )
    return STATUS_WAIT_1;

  pNtHeader = (PIMAGE_NT_HEADERS32)((DWORD)infosPE + ((PIMAGE_DOS_HEADER)
      infosPE)->e_lfanew);
  // hash of 'P' 'E' '0' '0' (0x00004550) => 0x0F750B7D4
  if ( (pNtHeader->Signature ^ 0xF750F284) != (IMAGE_NT_SIGNATURE ^ 0xF750F284)
      )
    return STATUS_WAIT_1;

  pFileHeader = (PIMAGE_FILE_HEADER)&pNtHeader->FileHeader;
  pOptionHeader = (PIMAGE_OPTIONAL_HEADER32)&((PIMAGE_NT_HEADERS32)infosPE)->
      OptionalHeader;
  if ((pFileHeader->Machine ^ 0x594F) == (IMAGE_PE_i386_MACHINE ^ 0x594F))
  {
    if ((pOptionHeader->Magic ^ 0x5908) == (IMAGE_PE32_MAGIC ^ 0x5908) && (
        pFileHeader->SizeOfOptionalHeader == 0xE0 ))
    {
      pPEData->Status = 0;
      pExportTableRVA = (PVOID)&(pOptionHeader->DataDirectory[
          IMAGE_DIRECTORY_ENTRY_EXPORT]).VirtualAddress;

Continue:
      pPEData->ExportTableRVA=pExportTableRVA;
      pPEData->dwSizeOfImage = pOptionHeader->SizeOfImage;
      pPEData->lpDataDir=pFileHeader->SizeOfOptionalHeader + (BYTE *)&(
          pFileHeader->Characteristics) + sizeof(WORD);
      pPEData->ResourceDataDir=(ULONG)infosPE;
      pPEData->PEAddress1=pPEData->PEAddress2;
      pPEData->wNumberOfSections=pFileHeader->NumberOfSections;
      pPEData->PEAddress2=(PIMAGE_DOS_HEADER)&((PIMAGE_NT_HEADERS32)infosPE)->
          Signature;
      return STATUS_SUCCESS;
     }
  }
  else if ((pFileHeader->Machine ^ 0xDE67) == (IMAGE_PE_x86_MACHINE ^ 0xDE67)
           && (pOptionHeader->Magic ^ 0x5A08) == IMAGE_PE32_PLUS_MAGIC
           && pFileHeader->SizeOfOptionalHeader == 0xF0 ) // 0x5803 64bits
  {
    pPEData->Status = 1;
    pExportTableRVA = (PVOID)&(pOptionHeader->DataDirectory[
        IMAGE_DIRECTORY_ENTRY_RESOURCE]).VirtualAddress;
    goto Continue;
  }
  return STATUS_WAIT_1;
}
```

Figure 3. Code source final de la fonction ParsePE

```
// int __usercall SearchForCodeInSystem<eax>(int *a1<edi>, int a2<esi>);
```

Une convention d'appel non standard est détectée dans le cas où une partie de la fonction a été écrite directement en assembleur ou à la suite d'une optimisation faite par le compilateur. On doit alors réécrire la fonction à la main (en partie en assembleur) ou choisir à la main une convention d'appel.

Convention	Arguments	*this* (C++)	Retour	Nettoie la pile
C (__cdelcl)	pile	(argument)	eax	appelant
Standard (__stdcall)	pile	(argument)	eax	appelé
Thiscall (__thiscall)	pile	ecx	eax	appelé
Fastcall (__fastcall)	ecx, edx, pile	(argument)	eax	appelé

Figure 4. Conventions d'appel dans leur version Visual C++

3 Analyse fonctionnelle du driver à partir de son code source

Une fois le code du driver reconstitué, décrivons en détail son fonctionnement. Il y a deux phases principales, la première consiste en la mise en place du driver : il demande au système d'être notifié en cas de chargement de binaires et initialise ses mécanismes de furtivité. La seconde phase est lancée lorsque des notifications sont signalées chargeant un binaire cible. Le driver infecte alors le binaire en y injectant la DLL de `Duqu` puis celle-ci active la charge finale.

3.1 Initialisation du driver lors du démarrage du système

La séquence de démarrage des drivers système sous Windows est déterminée par le positionnement de la clé de registre `Group`. Ainsi en fonction de son groupe d'appartenance (`nfrd965.sys` fait partie du groupe "network"), le driver sera démarré plus ou moins tôt lors de la séquence de boot. Le système d'exploitation donne automatiquement la main aux drivers prioritaires avant même que la couche d'abstraction matérielle (HAL) ne soit chargée en mémoire.

Le driver, une fois démarré, commence par allouer un emplacement mémoire de 512 octets destiné à contenir un tableau de pointeurs de fonctions partagées entre les différentes routines de "callback". A la suite de

cela, il passe au déchiffrement de ses paramètres internes (avec la fonction donnée en figure 5) nous révélant ainsi le nom et l'emplacement de la clé de registre utilisée pour la configuration de "l'injection" mais également le nom sous lequel le driver va s'enregistrer au près du gestionnaire.

```
 void __fastcall decode_parameters(
      __in PUNICODE_STRING RegistryPath)
{
  ULONG dwSeed;
  UINT32 dwCount;

  dwSeed = 0x7EF640F0;
  dwCount = 0;

  do
  {
    g_DefaultParamCrypted[dwCount++] ^= (char) dwSeed;
    dwSeed = ((((dwSeed & 0xFFFFFFFB) << 28) | (dwSeed >> 5)) * (((dwSeed & 0
        xFFFFFFFB) << 28)
    | (dwSeed >> 5)) / 0x8677 + 0x787C956A * (((dwSeed & 0xFFFFFFFB) << 28) | (
        dwSeed >> 5)) + 1)
    ^ (((dwSeed & 0xFFFFFFFB) << 28) | (dwSeed >> 5));
  } while (dwCount < 428 );

  if ( (char)g_LocalParameters == 0) {
      //Recopie le chemin de la cle de registre lorsque cela est necessaire
      memcpy((void *)&g_LocalParameters, RegistryPath->Buffer, RegistryPath->
          Length);
  }
}
```

Figure 5. Routine de déchiffrement utilisée pour les paramètres internes (également utilisée sur la DLL à injecter).

Si le déchiffrement s'est correctement déroulé, vient alors la vérification du mode d'exécution : soit le système s'avère être en mode sans échec ou en mode débogage, dans ce cas le driver termine son exécution ; soit en mode normal alors il commence par créer un device répondant au doux nom de `\Device\{624409B3-4CEF-41c0-8B81-7634279A41E5}` puis il définit la liste des commandes de contrôle qu'il sera à même de traiter (une grande majorité des commandes commencent par `IRP_MJ_` tels que `IRP_MJ_CREATE`, `IRP_MJ_READ`, etc.).

Ceci étant fait, il enregistre auprès du gestionnaire d'événements interne du noyau, deux fonctions de rappel (*callback function*). La première est nécessaire au gestionnaire du `Plug and Play` (`PnP`). On y trouve des opérations classiques pour un driver, à savoir la création d'un point d'ac-

cès sur celui-ci (\Device\Gpd0), ainsi qu'un lien (\DosDevices\GpdDev) et enfin l'attachement de l'objet Device à une pile mémoire. Quant à la seconde, elle sera exécutée lorsque le driver devra être initialisé ou réinitialisé (à la différence de la première, cette fonction est insérée dans une liste d'attente d'événements et supprimée à l'issue de son traitement).

Cette seconde fonction embarque un mécanisme d'attente de fin de chargement du noyau Windows. Pour cela, une fonction vérifie qu'il est possible d'accéder à la DLL hal.dll. Si tel n'est pas le cas, la fonction de réinitialisation est de nouveau insérée dans la file d'attente des traitements pour un maximum de 200 fois. Lors de la fin de chargement du système, un point d'accès \Device\Gpd1 est alors créé puis une routine de traitement des requêtes lui est associé. À ce stade le driver est à même de dialoguer avec une application s'exécutant en mode utilisateur (ring 3).

La mise en place des fonctions du rootkit va pouvoir débuter. On parle ici de rootkit car ce dispositif ne peut pas se permettre d'utiliser des appels systèmes réputés pour être couramment employées par les malwares et donc particulièrement surveillées par les antivirus. En particulier les fonctions d'allocation d'espace mémoire dans d'autres processus (tels VirtualAllocEx et ZwAllocateVirtualMemory), car elles permettent d'injecter du code dans un processus cible. Mais pour accomplir sa tâche, qui va consister à déposer un *HOOK* sur le point d'entrée d'une application système, Duqu a également besoin de faire appel à la fonction ZwProtectVirtualMemory que Microsoft a délibérément omise de la liste des fonctions accessibles en dehors du kernel. C'est pourquoi la stratégie de Duqu va consister à retrouver cette fonction en mémoire afin de réaliser des appels directs sans passer par le système standard des imports.

Ainsi le driver part à la recherche des fonctions ZwAllocateVirtualMemory et ZwProtectVirtualMemory afin de localiser leur emplacement mémoire tout en se préservant d'éventuelles détections. Le procédé que nous allons voir ensemble est remarquable d'ingéniosité car ces deux fonctions sont implémentées directement au cœur du noyau système (fichier Ntoskrnl.exe ou ntkrnlpa.exe suivant la version de l'OS).

Une première fonction se charge de passer en revue la liste de tous les modules chargés par le système de démarrage (DLLs et exécutables) jusqu'à identifier l'un des deux fichiers recherchés en comparant leurs noms. Cette liste est obtenue à l'aide de la fonction système ZwQuerySystemInformation.

Lorsque le fichier noyau est déniché, le processus prend connaissance de son emplacement mémoire afin de le soumettre à une seconde fonction qui va le disséquer.

Le fichier noyau est au format `PE` (`Portable Executable`). Sur la figure 3 précédemment évoquée, nous pouvons voir comment cette fonction procède pour s'assurer qu'il s'agit bien d'un fichier exécutable et pour déterminer l'emplacement des sections et de la table d'export. Elle réalise un contrôle sur les valeurs remarquables telles que 'MZ', 'IMAGE_NT_SIGNATURE', 'IMAGE_PE_i386_MACHINE, 'IMAGE_PE_x86_MACHINE', 'IMAGE_PE32_MAGIC' ou encore 'IMAGE_PE32_PLUS_MAGIC' et progresse dans le fichier jusqu'à atteindre la partie définissant les sections. Ces constantes sont bien entendu obfusquées (avec un *XOR*) afin de ne pas éveiller de soupçons.

Une fois arrivée aux sections, la fonction va rechercher celle contenant le code exécutable des fonctions systèmes (section nommée `.text` ou `PAGE` et de type `IMAGE_SCN_MEM_EXECUTE`, `IMAGE_SCN_MEM_READ`, `IMAGE_SCN_CNT_CODE`). Là encore, dans un esprit de furtivité, elle utilise une fonction de hachage sur le nom des sections rencontrées et compare le nom haché à la valeur désirée (voir la figure 6).

Maintenant que la section a été débusquée, une nouvelle fonction va se charger de localiser une fonction système faisant appel à `ZwProtectVirtualMemory` sous la forme d'un `call` (voir la figure 7), étape nécessaire car cette fonction n'est pas exportée par le noyau.

La fonction recherchée contient l'opcode `push 104h` (encodé en `68 04 01 00 00`) et réalise un `call` (encodé en `E8`) un peu plus loin dans le code. Sachant que cette fonction effectue au préalable un appel vers la fonction `ZwAllocateVirtualMemory`, la recherche remonte jusqu'à trouver cet appel (l'adresse du `call` est comparée à celle de la fonction `ZwAllocateVirtualMemory` exportée par le noyau), puis redescend au prochain `call` (à l'adresse `0x004ED1EA`) qui est l'appel vers `ZwProtectVirtualMemory` recherché.

Le processus vérifie dans un premier temps que l'adresse trouvée pour la fonction `ZwProtectVirtualMemory` est bien située dans une fonction du noyau pour s'assurer qu'elle n'est pas déroutée vers une fonction de surveillance. Dans un second temps, elle contrôle son intégrité par l'application d'un masque sur les 32 octets composant la routine avec un `AND` logique (voir figure 8). Ces vérifications sont également effectuées pour la fonction `ZwAllocateVirtualMemory`.

A présent, l'adresse de la fonction `ZwProtectVirtualMemory` peut être mémorisée pour une future utilisation discrète.

```
pSectionHeader=(PIMAGE_SECTION_HEADER)(pFirstSectionHeader+sizeof(
    IMAGE_SECTION_HEADER)*wIndex);
if ( ((ULONG)(((PIMAGE_SECTION_HEADER)((DWORD)pPEDataPtr->lpDataDir+sizeof(
    IMAGE_SECTION_HEADER)
      * wIndex))->Characteristics) & (IMAGE_SCN_MEM_EXECUTE| IMAGE_SCN_MEM_READ
      | IMAGE_SCN_MEM_DISCARDABLE | IMAGE_SCN_CNT_CODE)) == (
          IMAGE_SCN_MEM_EXECUTE
      | IMAGE_SCN_MEM_READ | IMAGE_SCN_CNT_CODE) )
{
  SecNameHash = HashOfSectionName((BYTE *)((DWORD)pPEDataPtr->lpDataDir
        + sizeof(IMAGE_SECTION_HEADER) * wIndex));

  if ( SecNameHash == 0x4B93B9A8 || SecNameHash == 0x1294971C ) // hachage de
      ".text" et "PAGE"
  {
    SectionPhysicalAddress = (ULONG)(pSectionHeader->SizeOfRawData);
    if ( (ULONG)(pSectionHeader->Misc.PhysicalAddress) < SectionPhysicalAddress
         )
    SectionPhysicalAddress = (ULONG)(pSectionHeader->Misc.PhysicalAddress);
    OrdinalValue[0] = 1;
    FindCode(pPEDataPtr, pBaseAddress + pSectionHeader->VirtualAddress,
        pBaseAddress
          + SectionPhysicalAddress + pSectionHeader->VirtualAddress, &
              OrdinalValue[1],
          (BYTE *)&Status);
    if ( OrdinalValue[0] )
      break;
  }
}
```

Figure 6. Extrait de la routine de traitement du noyau système.

```
PAGE:004ED1AD                   loc_4ED1AD: [...]
PAGE:004ED1BC 50                push    eax             ; BaseAddress
PAGE:004ED1BD 57                push    edi             ; ProcessHandle
PAGE:004ED1BE E8 19 8C F1 FF    call    ZwAllocateVirtualMemory
PAGE:004ED1C3 3B C3             cmp     eax, ebx
PAGE:004ED1C5 8B 4D FC          mov     ecx, [ebp+BaseAddress]
PAGE:004ED1C8 89 4E 0C          mov     [esi+0Ch], ecx
PAGE:004ED1CB 7C 2E             jl      short loc_4ED1FB
PAGE:004ED1CD 38 5D 0B          cmp     byte ptr [ebp+ProcessHandle+3], bl
PAGE:004ED1D0 74 27             jz      short loc_4ED1F9
PAGE:004ED1D2 8B 45 D0          mov     eax, [ebp+var_30]
PAGE:004ED1D5 89 45 F8          mov     [ebp+ProtectSize], eax
PAGE:004ED1D8 8D 45 F4          lea     eax, [ebp+OldProtect]
PAGE:004ED1DB 50                push    eax             ; OldProtect
PAGE:004ED1DC 68 04 01 00 00    push    104h            ; NewProtect
PAGE:004ED1E1 8D 45 F8          lea     eax, [ebp+ProtectSize]
PAGE:004ED1E4 50                push    eax             ; ProtectSize
PAGE:004ED1E5 8D 45 FC          lea     eax, [ebp+BaseAddress]
PAGE:004ED1E8 50                push    eax             ; BaseAddress
PAGE:004ED1E9 57                push    edi             ; ProcessHandle
PAGE:004ED1EA E8 93 96 F1 FF    call    ZwProtectVirtualMemory
PAGE:004ED1EF 3B C3             cmp     eax, ebx
```

Figure 7. Fonction faisant appel à `ZwProtectVirtualMemory`.

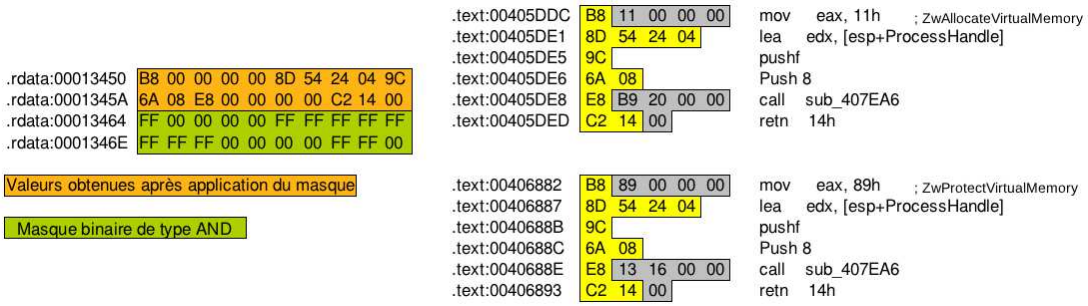

Figure 8. Application du masque d'intégrité sur les fonctions `ZwAllocateVirtualMemory` et `ZwProtectVirtualMemory`. Les valeurs qui doivent être identiques sont en gris clair.

Vient alors l'initialisation d'un espace mémoire partagé servant de lien entre les différentes fonctions de rappel au sein du noyau (ring 0). Il va servir à stocker, entre autres, les paramètres d'infection extraits depuis la base de registre une fois déchiffrés, mais également une structure (voir la figure 9) donnant accès à des fonctions système situées dans la DLL Kernel32.dll. Enfin, pour ses besoins interne, il va constituer une table d'import donnant accès à des fonctions situées au cœur du noyau (voir la figure 10). Cette table d'import pourra être utilisée par la suite par les différents codes injectés ainsi que la charge finale.

```
typedef struct _INJECTEDDATA
{
LONGLONG Kernel32Imagebase;       // Adresse de base de kernel32.dll
LONGLONG VirtualAlloc;            // Pointeurs sur les fonctions de Kernel32
    utilisees par le code injecte
LONGLONG VirtualFree;
LONGLONG GetProcAddress;
LONGLONG GetModuleHandleA;
LONGLONG LoadLibraryA;
LONGLONG LoadLibraryW;
LONGLONG lstrcmpA;
LONGLONG lstrcmpiA;
LONGLONG GetVersionExA;
LONGLONG DeviceIoControl;
LONGLONG InjectedSupModuleStart;  // Pointeur sur le debut de la memoire
    allouee dans le processus cible
LONGLONG InjectedSupModuleMZoffset; // Offset du debut du code tremplin injecte
LONGLONG InjectedSupModuleSize;   // Taille du code tremplin injecte
LONGLONG InjectedDLLImagebase;    // Pointeur sur le debut de la DLL injectee
ULONG EntryPoint1stDword;         // Sauvegarde des douze premiers octets du
    point d'entree de la cible
ULONG EntryPoint2snDword;
ULONG EntryPoint3rdDword;
ULONG Reserved15;
ULONG OldMemProtect;              // Sauvegarde des droits d'acces originaux
ULONG Reserved17;
HANDLE handle;                    // Sauvegarde de l'Handle permettant de
    communiquer avec le driver depuis l'espace utilisateur
ULONG Reserved18;
}_INJECTEDDATA, *PINJECTEDDATA;
```

Figure 9. Structure mémorisée dans un espace mémoire partagé.

La phase d'initialisation du driver se termine par la mise en place d'une notification système dès lors qu'une image d'un fichier exécutable est chargée ou mappée en mémoire (appel système `PsSetLoadImageNotifyRoutine`).

```
int *__cdecl InitializeIAT()
{
  PIMAGE_DOS_HEADER pBaseAddress;
  int KeUnstackDetachProcessTmp;
  struct _PEData pedata;
  PEDataPtr pPEData=NULL;

  pBaseAddress = SearchNtoskrnl();

  if ( pBaseAddress )
  {
    if ( !ParsePE((PEDataPtr)&pPEData, pBaseAddress,1) )
    {
      // Recherche de la fonction PsSetLoadImageNotifyRoutine()
      g_PsSetLoadImageNotifyRoutineADDR = GetApiFromKernel32((PEDataPtr)&
          pPEData, 0x0AA67394D);
      // Recherche de la fonction ZwAllocateVirtualMemory()
      g_ZwAllocateVirtualMemoryADDR     = GetApiFromKernel32((PEDataPtr)&
          pPEData, 0x6DFD9339);
      // Recherche de la fonction KeStackAttachProcess()
      g_KeStackAttachProcessADDR        = GetApiFromKernel32((PEDataPtr)&
          pPEData, 0x0A81306FD);
      // Recherche de la fonction KeUnstackDetachProcess()
      KeUnstackDetachProcessTmp       = GetApiFromKernel32((PEDataPtr)&pPEData,
           0x7AFE8755);
      g_KeUnstackDetachProcessADDR = KeUnstackDetachProcessTmp;
      if ( g_PsSetLoadImageNotifyRoutineADDR )
      {
        if ( g_ZwAllocateVirtualMemoryADDR && g_KeStackAttachProcessADDR
             && KeUnstackDetachProcessTmp && !IsWin2kSystem() )
        {
          // Recherche de la fonction RtlGetVersion()
          g_RtlGetVersionADDR         = GetApiFromKernel32((PEDataPtr)&pPEData,
               0x0CA5EF8B8);
          // Recherche de la fonction KeAreAllApcsDisabled()
          g_KeAreAllApcsDisabledADDR  = GetApiFromKernel32((PEDataPtr)&pPEData,
               0x0B08DC9B0);
          // Recherche de la fonction PsGetProcessSessionId()
          g_PsGetProcessSessionIdADDR = GetApiFromKernel32((PEDataPtr)&pPEData,
               0x885399FC);
          // Recherche de la fonction PsGetProcessPeb()
          g_PsGetProcessPebADDR       = GetApiFromKernel32((PEDataPtr)&pPEData,
               0x5A68953C);
        }
      }
    }
  }
  return &g_RtlGetVersionADDR;
}
```

Figure 10. Création de la table d'import des fonctions système issues du noyau.

3.2 Traitement de la première notification

Préparation à l'injection Au cours de son fonctionnement, le système d'exploitation va réaliser des opérations de chargement de modules en mémoire (DLLs ou fichiers exécutables). C'est à l'issue de ce processus que l'OS va redonner la main au driver en exécutant la routine de notification précédemment définie dans la phase d'initialisation du driver.

Le driver commence par vérifier qu'il se trouve bien sous une version de l'OS totalement supportée avant de poursuivre par la localisation du module fraîchement mappé en mémoire et pas encore démarré. Pour cela, il se sert de l'`id` du processus que lui a passé le système d'exploitation en paramètre. Il lit l'adresse de base du fichier directement à partir des informations accessibles dans la structure PEB (Process Enviroment Block) et la compare à celle passée en paramètre par le système. Il vérifie que le chemin d'accès des données de configuration a été préalablement déchiffré et par conséquent que la configuration est accessible en clair depuis la mémoire partagée. Il procède alors à la recherche de sa cible. Pour cela, il parcourt la liste des processus visés et détermine la charge à injecter. Malheureusement nous n'avons pas eu en notre possession le contenu de la clé de registre contenant la définition de la configuration. Nous nous sommes alors appuyés sur le document de `CrySys`, dans lequel apparaît une représentation pour le driver `JMINET7.SYS`.

Injection de la charge finale Une fois le processus cible repéré dans la liste, le driver vérifie qu'il n'y est pas déjà attaché. S'il y est attaché, il s'en détache (fonction système `KeUnstackDetachProcess`) et termine son exécution ; dans le cas contraire il continue l'offensive. À ce stade, il connaît l'emplacement de la charge sauvegardée sur le disque sous la forme d'un fichier chiffré. Une fonction va donc lire ce fichier en mémoire puis le déchiffrer à l'aide du même algorithme utilisé sur les paramètres internes. Elle mémorise la position de la DLL ainsi que sa taille dans une structure interne.

Selon le fichier de configuration, plusieurs binaires peuvent être ciblés pour l'injection. Par défaut, `Duqu` choisit `services.exe`, nous nous sommes restreints à l'analyse de l'injection dans ce cas mais celle-ci serait similaire pour un autre binaire. Pour commencer, une fonction scrute l'entête du fichier `services.exe` à la recherche de son point d'entrée, puis alloue un espace mémoire pour y transférer une portion de code présent dans la section `.data` du driver. Pour ne pas alarmer un dispositif de surveillance tel qu'un antivirus, ce morceau de code s'avère être un fichier exécutable dont l'entête a

été altéré. La suite du traitement va donc rétablir les constantes manquantes ('MZ', 'IMAGE_NT_SIGNATURE', 'IMAGE_PE_i386_MACHINE, et 'IMAGE_PE32_MAGIC'). Elle termine par la résolution des éventuels problèmes de relocation d'adresse et enfin sauvegarde les droits d'accès à la page mémoire contenant le point d'entrée, avant de les changer de RX (PAGE_EXECUTE_READ) en RWX (PAGE_EXECUTE_WRITECOPY) en utilisant la fonction système ZwProtectVirtualMemory ; voilà pourquoi les développeurs se sont donnés du mal à localiser un appel direct à cette fonction.

Il est temps de commencer l'intrusion. Une nouvelle fonction alloue un emplacement dans l'espace mémoire du processus services.exe de la taille de la DLL augmentée de 57 octets et y copie la fameuse charge. S'en suit alors l'ouverture d'un *handler* sur le driver noyau (nfrd965.sys), sauvegardé dans une structure afin de le communiquer ultérieurement à la portion de code injecté. La routine de notification se termine par le détachement du processus services.exe (fonction système KeUnstackDetachProcess) et libère quelques ressources internes.

3.3 Traitement de la seconde notification

A la réception d'une nouvelle notification, la routine de traitement examine si le module chargé correspond à la DLL kernel32.dll. Dans ce cas, elle regarde si le driver est déjà attaché au processus qui a provoqué la notification, si ce n'est pas le cas elle l'attache (fonction système KeStackAttachProcess). Vient alors la phase de recherche des 10 fonctions exportées par kernel32.dll nécessaires au code injecté dans services.exe. Comme toujours, la recherche se doit d'être discrète, elle consiste à comparer le hash des noms des fonctions désirées. Là aussi, la routine de notification se termine par le détachement du processus services.exe mais avant elle procède à la sauvegarde des 12 premier octets composant le code du point d'entrée avant de les remplacer par un *mov eax,@adresseInjection* suivi d'un *call eax*.

Le mécanisme de lancement de la charge est maintenant armé. Laissons faire le système.

3.4 Exécution de la charge

Le système d'exploitation termine l'initialisation du processus services.exe puis procède à son exécution en donnant la main au code indiqué par son point d'entrée. En réalité, le système ne fait que démarrer le code injecté dans ce processus à son insu. La première étape du code

injecté consiste à déterminer sa propre adresse en mémoire car toutes ses futures opérations en dépendent (lecture, écriture et saut en mémoire). Le procédé est classique et un exemple est donné Figure 11 : il suffit de réaliser un saut à l'instruction suivante (*call +5*), ce qui a pour effet de placer l'adresse de retour (où est présente l'instruction *pop eax*) en haut de la pile, puis de dépiler l'adresse de retour dans un registre (ici dans `eax` : *pop eax*). Lors de la compilation, les adresses de chaque cible de saut sont calculées par valeur relative, par rapport au point d'entrée du code injecté (*offset*). Lors de l'exécution, on peut déterminer l'adresse courante absolue, puis lui soustraire l'adresse relative de l'instruction où se fait ce calcul (déterminée à la compilation). On obtient alors l'adresse absolue de début du code injecté. Il suffit enfin d'y ajouter l'adresse relative de la cible du sauf pour recalculer l'adresse absolue correspondante.

```
call $+5
pop  eax
; eax = adresse absolue courante de l'instruction "pop eax"
lea edx, ds:0x6E2
; edx = adresse relative de "pop eax" par rapport au debut du code injecte
sub eax, edx
; eax = adresse absolue du debut du code injecte
; ecx = offset : adresse relative recherchee
add eax, ecx
; eax = adresse absolue recherchee
```

Figure 11. Technique utilisée pour déterminer son adresse absolue.

Il restaure les constantes de l'entête du second code injecté afin qu'il soit conforme au format `PE`, puis, renseigne une structure à partir des informations stockées précédemment (les 10 fonctions repérées dans la DLL `kernel32.dll`) afin de constituer une table d'import (fonctions `GetModuleHandleA`, `MemAlloc`, `MemFree`, `LoadLibraryA`, `GetProcAddress` et `GetVersionExA`). Ensuite il appelle la fonction `GetModuleHandleA` pour obtenir un *handler* sur la DLL `ntdll.dll` qu'il stocke également dans cette même structure. Il saute alors au point d'entrée du second code injecté en lui passant l'adresse de deux structures (dont la table d'import fraîchement renseignée).

La première opération réalisée par ce second module supplémentaire, consiste à vérifier que l'entête de la structure reçue en paramètre commence bien par la valeur 1539. Ensuite il remplie la seconde structure avec les informations issues de son propre entête PE (adresse de *mappage*

du module, taille, nombre de sections et adresse de la table d'export). Enfin, une nouvelle fonction récupère ces informations et les utilise pour *mapper* le fichier en mémoire comme l'aurait fait le système d'exploitation (allocation d'un espace mémoire, copie de l'entête PE, mappage des sections, chargement des DLL, création de la table d'import, correction des adresses mentionnées dans la table de relocalisation et enfin calcul du point d'entrée). A l'issue de toutes ces opérations, la routine appelle le nouveau module re-localisé par son véritable point d'entrée. Cette fois-ci, elle va s'affairer à re-localiser la DLL `NETP191.PNF` puis d'appeler son point d'entrée.

Lorsque le code injecté termine son action offensive, il envoie une première requête au driver à travers le point d'accès `\Device\{624409B3-4CEF-41c0-8B81-7634279A41E5}` afin que celui-ci modifie les droits d'accès de la page mémoire au point d'entrée de `services.exe`. Ainsi il peut rétablir les 12 octets préalablement sauvegardés par le driver pour lui dans la seconde structure (voir la figure 9). Il termine sa tâche par une seconde requête afin, cette fois-ci, de restaurer les droits d'accès d'origine.

La mission de l'injection étant atteinte, l'exécution reprend son cours et cette fois-ci le processus `services.exe` original est exécuté.

4 Utilisation du driver

Nous avons, dans la partie précédente, détaillé le déroulement de l'injection de la DLL de `Duqu` dans `services.exe`. Certains de ces mécanismes, comme la mise en place de notifications lors du chargement en mémoire d'un binaire, peuvent clairement être utilisées à des fins défensives.

Schématiquement, le driver modifié calcule des signatures sur les binaires chargés en mémoire. Lorsqu'un binaire est lancé, dans le cas où sa signature n'est pas reconnue, un scan de la mémoire lui étant allouée est effectué, pouvant révéler la présence d'un malware. Nous allons à présent détailler les phases d'initialisation, de mémorisation puis de détection du driver détourné. Nous terminerons ce papier par une présentation détaillée d'une exécution du binaire modifié en situation d'attaque par `Duqu`.

4.1 Phase d'initialisation

La phase d'initialisation de notre driver a été grandement allégée. Nous avons gardé la partie commune à tout driver à savoir la création des

points d'accès (en prenant soin de les renommer) et supprimé la recherche de la fonction `ZwProtectVirtualMemory`. Par souci de simplification, les paramètres internes n'ont pas été chiffrés (chemin de la clé de registre `FILTER` et nom du device). Nous avons bien entendu conservé le système de notification des chargements de modules mais en plus nous lui avons ajouté une autre notification : lorsque le système termine la création d'un processus (fonction `PsSetCreateProcessNotifyRoutine`).

4.2 Phase de mémorisation

Nous avons vu précédemment que `Duqu` écrase le code du point d'entrée du service infecté uniquement à l'instant où `kernel32.dll` vient d'être mappée en mémoire. Nous avons donc tout le temps de réaliser une somme de contrôle à chaque point d'entrée avant son altération. Nous avons repris intégralement la fonction de hachage implémentée dans `Duqu` et utilisée pour retrouver le nom des fonctions de façon discrète (voir la figure 12).

```
ULONG __fastcall HashOfSectionName(
      __inout BYTE *byte)
{
  int step1;
  int step2;
  int step3;
  int step4;
  int step5;
  int step6;

  DbgPrint("Entrypoint bytes at 0x%08x: 0x%02x 0x%02x 0x%02x 0x%02x 0x%02x 0x
      %02x 0x%02x 0x%02x\n",
            byte,byte[0],byte[1],byte[2],byte[3],byte[4],byte[5],byte[6],byte
                [7]);

  step1 = byte[1] + byte[0] * (5 * byte[0] + 9);
  step2 = byte[2] + step1 * (5 * step1 + 9);
  step3 = byte[3] + step2 * (5 * step2 + 9);
  step4 = byte[4] + step3 * (5 * step3 + 9);
  step5 = byte[5] + step4 * (5 * step4 + 9);
  step6 = byte[6] + step5 * (5 * step5 + 9);
  return  byte[7] + step6 * (5 * step6 + 9);
}
```

Figure 12. Fonction de hachage sur un tableau de 8 octets.

La première notification surviendra lorsqu'un nouveau processus sera créé par le système. Malheureusement pour nous, le système d'exploitation

nous notifie uniquement l'`id` du processus, son parent et s'il vient d'être créé ou détruit. Cela dit, Windows peut nous fournir la structure PEB (Process Enviroment Block) associée au processus. Nous l'avons utilisée afin de déterminer l'adresse mémoire du fichier.

Pour une raison de simplification, nous allons nous focaliser uniquement sur l'infection du processus `services.exe` (visé par `Duqu`). C'est pourquoi nous allons interroger le système afin de récupérer le nom du processus et de le comparer à la chaîne de caractères `services.exe`. Si le processus est identifié comme étant `services.exe`, nous procédons à la recherche de son point d'entrée, nous calculons son hash puis le sauvegardons comme valeur de référence. Nous sommes maintenant prêts à détecter la mise en place du *hook* réalisé par `Duqu`.

4.3 Phase de détection

Lorsque le système d'exploitation redonne la main à notre routine à la suite du chargement d'un module en mémoire, celle-ci commence par mémoriser l'adresse de base du module notifié puis l'utilise ensuite pour scruter l'entête du fichier `PE` à la recherche de son point d'entrée. Nous nous sommes contentés de reprendre ce qui a été fait dans `Duqu` en ajoutant simplement le calcul de la somme de contrôle.

Passons au calcul du contrôle d'intégrité du processus surveillé (`services.exe`). Dans la phase précédente, nous avons sauvegardé son point d'entrée ainsi qu'un hash de référence. Nous procédons à un nouveau calcul de ce hash et le comparons à celui de référence. Si les deux hash sont différents, nous en déduisons que `Duqu` a positionné son *hook* entre deux notifications. Nous pouvons ainsi lire l'adresse cible du saut, procéder à un scan mémoire et analyser le code du processus ayant subi l'injection.

4.4 Démonstration

Pour faciliter la mise au point de ce driver de détection, nous l'avons installé manuellement sur une machine de test ainsi que celui de `Duqu` en nous inspirant fortement de la démarche proposée par Sergei Shevchenko [8]. Ainsi lorsque nous ouvrons la calculatrice renommée en `services.exe`, notre driver entre en action.

Cependant, lors de la reconstitution du code source, nous avons remarqué que le driver `nfrd965.sys` vérifie que le système n'est ni en mode sans échec, ni en mode débogage (Figure 13).

```
if(status == STATUS_SUCCESS) {
   decode_parameters(RegistryPath);
   if ( ((ULONG)&g_DefaultParamCrypted & 1) && *InitSafeBootMode != 0) {
                status = STATUS_UNSUCCESSFUL;
   } else if ( ((ULONG)&g_DefaultParamCrypted & 2)) {
             if (*KdDebuggerEnabled ==0 ) {
                   status = STATUS_UNSUCCESSFUL;
             } else {
                   status = STATUS_SUCCESS;
             }
   } else {
             status = STATUS_SUCCESS;
   }
```

Figure 13. Détection du mode sans échec, ainsi que le mode débogage.

Pour retirer cette restriction nous avons patché le driver comme le montre la Figure 14 : on remplace simplement un saut conditionnel *jz* par un saut inconditionnel *jmp*. Nous copions ce driver ainsi patché dans le répertoire `C\WINDOWS\system32\drivers\` ainsi que la DLL chiffrée à l'aide de la routine extraite du code source dans le répertoire `C\WINDOWS\inf\` sous le nom `NETP191.PNF`. Il nous reste à configurer les clés de registre renfermant les paramètres utilisés par le driver lors de son initialisation. Les valeurs remarquables sont le nom du driver (`DisplayName=nfrd965`), son groupe (`Group=Network`), son emplacement (`ImagePath`=C :\WINDOWS\System32 \Drivers\nfrd965.sys), la valeur de ses paramètres internes (repris de `Crysys`) et les paramètres de démarrage (`Start=3` et `Type=1`). L'attribut `Type` indique qu'il s'agit d'un driver du noyau pour un appareil (ici de type `Network`), l'attribut `Start` spécifie le moment de lancement du driver : nous l'avons positionné à 3 (lancement à la demande) pour nous permettre de démarrer le driver défensif avant ou après celui de `Duqu`.

```
.text:00010611 loc_10611:                              ; CODE XREF: DriverEntry+   ; CODE XRE
.text:00010611                                         ; DriverEntry+98j           ; DriverEr
.text:00010611                mov     edx, dword_15190                 mov     edx, dword_15190
.text:00010617                test    edx, 2                           test    edx, 2
.text:0001061D                jz      short loc_10630                  jz      short loc_10630
.text:0001061F                mov     eax, ds:KdDebuggerEnabled        mov     eax, ds:KdDebuggerEnabled
.text:00010624                cmp     byte ptr [eax], 0                cmp     byte ptr [eax], 0
.text:00010627                jz      short loc_10630                  jmp     short loc_10630
.text:00010629                mov     eax, 0C0000001h                  mov     eax, 0C0000001h
.text:0001062E                jmp     short loc_10632                  jmp     short loc_10632
```

Figure 14. Patch de la détection du mode débogage.

Lors de cette démonstration nous utilisons deux machines virtuelles sous `Windows XPSP3` reliées par un cable série lui aussi virtuel. Sur la première machine, nous lançons le débogueur de Microsoft `WinDBG` en mode `kernel` écoutant sur le port `COM2`. Sur la seconde, le système est démarré en mode `"kernel debug"`. Cette configuration nous permettra de lancer notre driver de détection puis le driver `nfrd965.sys` de `Duqu` ou inversement.

En regardant de près la fenêtre de commandes du débogueur, nous constatons le chargement en mémoire du driver de `Duqu` car le système a envoyé une notification à notre driver à la fin de sa tâche (voir la Figure 15).

```
--** Load driver \??\c:\WINDOWS\system32\drivers\nfrd965.sys **--
LoadImageNotifyRoutine: ImageBaseAddress=0xf78c5000 ProcessId=0x0
  ParsePEModule: MZ
  ParsePEModule: PE
  ParsePEModule: Machine
Entrypoint bytes at 0xf78c5570: 0x55 0x8b 0xec 0x6a 0xff 0x68 0x08 0xa1
  ParsePEModule: DataDirectory=0xf78c5160 EntryPoint=0xf78c5570
LoadImageNotifyRoutine: Entrypoint=0xf78c5570 DataDirectory=0xf78c5160
```

Figure 15. Exécution du driver nfrd965.sys. La routine `LoadImageNotifyRoutine` de notre driver nous le signale.

Pour déterminer plus facilement le moment où intervient notre driver, nous décidons de positionner deux points d'arrêt dans le driver `Duqu` . Le premier, au moment où il sauvegarde les octets du point d'entrée et le second au moment où il exécute le code du module tremplin. Nous pouvons dès lors lancer `services.exe` : la réaction est immédiate (voir la figure 16).

La logique voudrait que le système commence par charger le fichier `services.exe` en mémoire puis crée un processus du même nom. Mais on constate que notre driver reçoit les notifications dans l'ordre inverse. Heureusement cela n'a pas d'impact sur notre méthode de détection. Ainsi lors de la création du processus, nous sauvegardons son `id`, son point d'entrée et calculons la somme de contrôle qui servira de signature.

Nous poursuivons son exécution et arrivons à l'instant où le système charge la DLL `kernel.dll` en mémoire et par conséquence notifie le driver `nfrd965.sys` (voir la Figure 17).

Lorsque notre driver est démarré avant le driver `nfrd965.sys`, nous recevons la notification de chargement de module avant lui. Ainsi lorsque

```
----------++++++++++++++++ Create process 0x914 ****************----------
ProcessImageInformation: PEB=0x7ffd6000 ImageBaseAddress=0x01000000
    UniqueProcessId=0x914
Entrypoint bytes at 0x01012475: 0x6a 0x70 0x68 0xe0 0x15 0x00 0x01 0xe8
  ParsePEModule: DataDirectory=0x01000168 EntryPoint=0x01012475
ProcessImageInformation: Entrypoint=0x01012475 DataDirectory=0x01000168
ProcessImageName: \Device\HarddiskVolume1\Documents and Settings\fabrice\Bureau
    \services.exe
======= ProcessImageName: save processID=0x914
CreateProcessNotify: ImageBaseAddress=0x01000000 EntryPoint=0x01012475
CreateProcessNotify: Create=1 0x788=>0x914
CreateProcessNotify: EntrypointChecksum=0x49af1bf2
--** Load module \Device\HarddiskVolume1\Documents and Settings\fabrice\Bureau\
    services.exe **--
LoadImageNotifyRoutine: ImageBaseAddress=0x01000000 ProcessId=0x914
Entrypoint bytes at 0x01012475: 0x6a 0x70 0x68 0xe0 0x15 0x00 0x01 0xe8
  ParsePEModule: DataDirectory=0x01000168 EntryPoint=0x01012475
LoadImageNotifyRoutine: Entrypoint=0x01012475 DataDirectory=0x01000168
------>>> Verify services.exe processus: Entrypoint bytes at 0x01012475: 0x6a 0
    x70 0x68 0xe0 0x15 0x00 0x01 0xe8
 OK!
```

Figure 16. Vue depuis la console de WinDbg : Exécution du programme `services.exe`. Notre driver le détecte et procède à l'enregistrement de son `id` (`0x914`), son point d'entrée (`0x01012475`) et de son hash (`0x49af1bf2`).

```
------*** Load module \WINDOWS\system32\kernel32.dll ***---------
LoadImageNotifyRoutine: ImageBaseAddress=0x7c800000 ProcessId=0x914
Entrypoint bytes at 0x7c80b64e: 0x8b 0xff 0x55 0x8b 0xec 0x83 0x7d 0x0c
  ParsePEModule: DataDirectory=0x7c800168 EntryPoint=0x7c80b64e
LoadImageNotifyRoutine: Entrypoint=0x7c80b64e DataDirectory=0x7c800168
------>>> Verify services.exe processus:Entrypoint bytes at 0x01012475: 0x6a 0
    x70 0x68 0xe0 0x15 0x00 0x01 0xe8
 OK!
------**** Load module \WINDOWS\system32\shell32.dll ****------------
LoadImageNotifyRoutine: ImageBaseAddress=0x7c9d0000 ProcessId=0x914
Entrypoint bytes at 0x7c9f7496: 0x8b 0xff 0x55 0x8b 0xec 0x53 0x8b 0x5d
  ParsePEModule: DataDirectory=0x7c9d0158 EntryPoint=0x7c9f7496
LoadImageNotifyRoutine: Entrypoint=0x7c9f7496 DataDirectory=0x7c9d0158
------>>> Verify services.exe processus:Entrypoint bytes at 0x01012475: 0xb8 0
    xbd 0x18 0x0a 0x00 0xff 0xd0 0xe8

!!!!!!!!!!!!!!!!!!!!!!!!>>> Checksum error !!!!!!!!!!!!!!!!!!!!!!!!
DLL found at 0xb0028, waiting for analysis.
maindll.decrypted (Stuxnet) detected.
Terminating services.exe.
```

Figure 17. Détection de l'altération du point d'entrée (`0x01012475`) de `services.exe`.

la DLL `kernel32.dll` est notifiée à notre driver, `Duqu` n'a pas encore altéré le point d'entrée. C'est pourquoi la somme de contrôle est toujours correcte (cela n'aurait pas été le cas si `Duqu` avait démarré avant le notre). Par contre, au chargement de la DLL suivante (`shell32.dll`), nous détectons la modification du point d'entrée. Cette modification est déjà suspecte et pourrait justifier l'arrêt du processus `services.exe` mais nous voulons être capable de retrouver la similarité avec `Stuxnet` trouvée dans leurs DLLs. Une analyse manuelle du code injecté dans `services.exe` révèle que la DLL de `Duqu` est chargée à l'adresse `0xb0028`. Afin de retrouver automatiquement cette DLL, on peut tenter de suivre le code injecté (par analyse statique), scanner les pages mémoires allouées par `services.exe` ou scanner l'ensemble de la mémoire à la recherche d'images exécutables. Pour l'instant le driver se contente de scanner l'adresse trouvée à la main.

Nous connaissons à présent l'adresse où est situé le code injecté par `Duqu` dans `services.exe`. Le driver réalise alors le dump de cette DLL inconnue et, connaissant `Stuxnet`, un scan lève une alerte : elle présente des similarités avec ce dernier. Le driver met alors fin au processus utilisant la DLL (`services.exe`).

Notre méthode de détection fonctionne par comparaison des graphes des flot de contrôle (CFG) [4], plus de détails sur cette technique de détection et son application à `Duqu` et `Stuxnet` ont été donnés dans de précédents travaux [11,5].

5 Conclusion

Les similarités entre `Duqu` et `Stuxnet` nous ont poussé à chercher une technique capable de détecter `Duqu` dans une situation d'attaque. À partir de la DLL principale de `Stuxnet` nous reconnaissions celle de `Duqu` mais encore fallait-il avoir accès à celle-ci une fois déchiffrée. Nous avons détaillé la technique d'infection de `Duqu` et le fonctionnement de son driver, capable d'injecter discrètement du code dans `services.exe` à l'aide de routines du noyau. Nous avons donc reconstruit le code source de ce dernier et l'avons détourné à des fins défensives. Utilisé pour intercepter les chargement de binaires par le système d'exploitation et interfacé avec notre détecteur, le driver a été capable de détecter l'injection effectuée par `Duqu`. Il est de plus possible de classer l'attaque comme liée à `Stuxnet` à condition de fournir l'adresse où est présente la DLL injectée en mémoire.

Références

1. CVE-2011-3402. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-3402.
2. Hex-Rays Decompiler. http://www.hex-rays.com/products/decompiler/index.shtml.
3. How To Control Device Driver Load Order. http://support.microsoft.com/kb/115486.
4. Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. Architecture of a Morphological Malware Detector. *Journal in Computer Virology*, 5 :263–270, 2009.
5. Guillaume Bonfante, Jean-Yves Marion, Fabrice Sabatier, and Aurélien Thierry. Code synchronization by morphological analysis. *7th International Conference on Malicious and Unwanted Software (Malware 2012)*, October 2012.
6. Laboratory of Cryptography of Systems Security (CrySyS). Duqu : A Stuxnet-like malware found in the wild, October 2011.
7. Aleks (Kaspersky). The mystery of Duqu : Part Ten, March 2012. https://www.securelist.com/en/blog/208193425/The_mystery_of_Duqu_Part_Ten.
8. Sergei Shevchenko. Actually, my name is Duqu - Stuxnet is my middle name. http://baesystemsdetica.blogspot.fr/2012/03/actually-my-name-is-duqu-stuxnet-is-my_4108.html.
9. Symantec. W32.Duqu : The Precursor to the Next Stuxnet, October 2011.
10. Amr Thabet. Reversing Stuxnet's Rootkit (MRxNet) Into C++, January 2011. http://amrthabet.blogspot.fr/2011/01/reversing-stuxnets-rootkit-mrxnet-into.html.
11. Aurelien Thierry, Guillaume Bonfante, Joan Calvet, Jean-Yves Marion, and Fabrice Sabatier. Recognition of binary patterns by Morphological analysis. *REcon*, 2012.